Chapter 2. The SOAP Message

All SOAP messages are packaged in an XML document called an *envelope*, which is a structured container that holds one SOAP message. The metaphor is appropriate because you stuff everything you need to perform an operation into an envelope and send it to a recipient, who opens the envelope and reconstructs the original contents so that it can perform the operation you requested. The contents of the SOAP envelope conform to the SOAP specification,[1] allowing the sender and the recipient to exchange messages in a language-neutral way: for example, the sender can be written in Python and the recipient can be written in Java or C#. Neither side cares how the other side is implemented because they agree on how to interpret the envelope. In this chapter we'll get inside the SOAP envelope.

2.1 The HTTP Binding

The SOAP specification requires a SOAP implementation to support the transporting of SOAP XML payloads using HTTP, so it's no coincidence that the existing SOAP implementations all support HTTP. Of course, implementations are free to support any other transports as well, even though the spec doesn't describe them. There's nothing whatsoever about the SOAP payload that prohibits transporting messages over transports like SMTP, FTP, JMS, or even proprietary schemes; in fact, alternative transports are frequently discussed, and a few have been implemented. Nevertheless, since HTTP is the most prevalent SOAP transport to date, that's where we'll concentrate. Once you have a grasp of how SOAP binds to HTTP, you should be able to easily migrate your understanding to other transport mechanisms.

SOAP can certainly be used for request/response message exchanges like RPC, as well as inherently one-way exchanges like SMTP. The majority of Java-based SOAP implementations to date have implemented RPC-style messages, so that's where we'll spend most of our time; HTTP is a natural for an RPC-style exchange because it allows the request and response to occur as integral parts of a single transaction. However, one-way messaging shouldn't be overlooked, and nothing about HTTP prevents such an exchange. We'll look at one-way messaging in Chapter 8.

2.2 HTTP Request

The first SOAP message we'll look at is an RPC request. Although it's rather simple, it contains all of the elements required for a fully compliant SOAP message using an HTTP transport. The XML payload of the message is contained in an HTTP POST request. Take a quick look, but don't get too caught up in figuring out the details just yet. The following message asks the server to return the current temperature in degrees Celsius at the server's location:

[1] The spec can be found at http://www.w3.org/TR/SOAP. The SOAP 1.1 specification is not a W3C standard, but the SOAP 1.2 spec currently under development will be.

```
POST /LocalWeather HTTP/1.0
Host: www.mindstrm.com
Content-Type: text/xml; charset="utf-8"
Content-Length: 328
SOAPAction: "WeatherStation"

<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
    <m:GetCurrentTemperature xmlns:m="WeatherStation">
         <m:scale>Celsius</m:scale>
    </m:GetCurrentTemperature>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The SOAP HTTP request uses the HTTP POST method. Although a SOAP payload could be transported using some other method such as an HTTP GET, the HTTP binding defined in the SOAP specification requires the use of the POST method. The POST also specifies the name of the service being accessed. In the example, we're sending the data to `/LocalWeather` at the host specified later in the HTTP header. This tells the server how to route the request within its own processing space. Finally, our example indicates that we're using HTTP Version 1.0, although SOAP doesn't require a particular version of HTTP.

The `Host:` header field specifies the address of the server to which we're sending this request, *www.mindstrm.com*. The next header field, `Content-Type:`, tells the server that we're sending data using the `text/xml` media type. All SOAP messages must be sent using `text/xml`. The content type in the example also specifies that the data is encoded using the UTF-8 character set. The SOAP standard doesn't require any particular encoding. `Content-Length:` tells the server the character count of the POSTed SOAP XML payload data to follow.

So far, all the headers have been standard HTTP headers that apply to any HTTP POST messages. The next one, however, is SOAP specific. The `SOAPAction:` header field is required for all SOAP request messages transported using HTTP.[2] It provides some information to the HTTP server in the form of a URI that indicates the intent of the message. This information is contained in an HTTP header field rather than in the message itself because it doesn't require the system to process the XML payload first. In turn, this means that the server can determine if it does not have the information or resources necessary to process the request without actually parsing the message. Although this field can contain data in any format or even be empty, the field itself must be present. The header `SOAPAction: "WeatherStation"` could indicate that our request requires an active connection to the weather station located on the roof of the building where the server resides. If the server knows that the weather station has fallen off the building and was subsequently crushed by a passing car, it can respond without bothering to process the SOAP payload. This may not be a common scenario, but the point is that the server can use the URI specified in the `SOAPAction:` field to gain some insight into the intent of the message, and act accordingly. It's also important to know that the URI need not take any particular form. It can be a URL, a name, a word, or even a number, as long as it has meaning to the server that receives the message.

[2] This is a requirement in SOAP 1.1, but is expected to be optional in SOAP 1.2.

If the `SOAPAction:` field contains an empty string (`""`), then the intent of the message is actually being provided by the HTTP request URI, which in the example is `/LocalWeather`. The server may interpret this URI to mean that it should access the weather station, or it might have some other meaning. If the field contains no data at all, then the message contains no information about the meaning or intent of the enclosed message. In that case, we'd expect the server to go ahead and process the XML payload.

2.3 HTTP Response

An RPC-style request message usually results in a corresponding HTTP response. Of course, if the server can't get past the information in the HTTP headers, it can reply with an HTTP error of some kind. But assuming that the headers are processed correctly, the system is expected to respond with a SOAP response. Here's the HTTP response to the RPC-style request from the previous example:

```
HTTP/1.0 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: 359

<SOAP-ENV:Envelope
     xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
     SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
     <m:GetCurrentTemperatureResponse xmlns:m="WeatherStation">
           <m:temperature>26.6</m:temperature>
     </m:GetCurrentTemperatureResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The response's HTTP header fields are, for the most part, similar to those of the request. The response code of 200 in the first line of the header indicates that the server was able to process the SOAP XML payload. The `Content-Type:` and `Content-Length:` fields have the same meanings as they did in the request message. No other HTTP header fields are needed; the correlation between the request and response is implied by the fact that the HTTP POST is inherently a request/response mechanism. You send the request and get the response as part of a single transaction.

Let's change the original request message so that the scale reads as follows:

```
<m:scale>Calcium</m:scale>
```

You know that one, right? No, there is no Calcium scale for temperature; we've constructed an erroneous request. So we go ahead and send the SOAP request to the server. Assuming the weather station hasn't really fallen off the building, the server processes the request. As we expected, our SOAP processing code does not understand the Calcium temperature scale, and generates the following error response:

```
HTTP/1.0 500 Internal Server Error
Content-Type: text/xml; charset="utf-8"
Content-Length: 525

<SOAP-ENV:Envelope
          xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   <SOAP-ENV:Body>
      <SOAP-ENV:Fault>
         <faultcode>SOAP-ENV:Client</faultcode>
         <faultstring>Client Error</faultstring>
         <detail>
            <m:weatherfaultdetails xmlns:m="WeatherStation">
               <message>No such temperature scale: Calcium</message>
               <errorcode>1234</errorcode>
            </m:weatherfaultdetails>
         </detail>
      </SOAP-ENV:Fault>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The HTTP error code of 500 with the explanation "Internal Server Error" shows that an error occurred. Just as before, the header includes the `Content-Type:` and `Content-Length:` HTTP header fields. The SOAP spec says that if the message is received and understood, the response should be sent using a 2xx status code. When the server does not understand the message, or if the message is improperly formatted, is missing information, or cannot be processed for any other reason, the server must use HTTP code 500 ("Internal Server Error"). That HTTP header is followed by a SOAP envelope, which includes its own fault code, fault string, and a detailed description of the error.

I'm not convinced that this is the best way to handle SOAP faults, or that using a transport protocol error is really a good way to package a SOAP error. Transport errors and SOAP errors really don't have anything to do with each other. After all, the response includes a proper SOAP message that describes the error in detail. However, in fairness to the developers of the SOAP spec, this error-handling procedure is in line with the way HTTP delivers errors in retrieving, say, HTML documents. I could still argue both sides, but that's the requirement and it certainly doesn't interfere with the SOAP processing.

2.3.1 Summarizing the HTTP Binding

That's just about all we need to cover about the SOAP HTTP binding. The beauty of it, really, is that HTTP is a well-established mechanism that works extremely well for transporting XML, and therefore it's a good way to transport SOAP messages.

Some of the details of the XML in the previous examples might be obvious to you, even though we haven't covered them yet. If you understand the XML, great. Otherwise, fear not. The purpose of the discussion so far has been to show you what the HTTP binding for SOAP looks like. In the next section we'll take a detailed look at the SOAP envelope, which is the XML payload that contains the SOAP requests and responses.

2.4 The SOAP Envelope

The SOAP envelope represents the entirety of the XML for a SOAP request or response.[3] The `Envelope` is the highest-level XML element in the message, and it must be present for the message to be considered valid. So in essence, the `Envelope` represents the XML document that contains the SOAP message. The `Envelope` can contain an optional `Header` element that, if present, has to be the first subelement of the `Envelope`. The `Envelope` must contain a `Body` element.[4] If the `Envelope` contains a `Header` element, then the `Body` element has to come right after the `Header`; otherwise the `Body` has to be the first subelement of the `Envelope`.

A SOAP envelope packaged and transported using HTTP is similar to a paper envelope sent using the postal service. The SOAP envelope is the paper envelope; the SOAP header (if present) and the SOAP body are the contents of the paper envelope; and the HTTP headers are the physical address information on the outside of the paper envelope.

With a little imagination we can complete the analogy. The return address on a paper envelope has no direct counterpart in the SOAP process; however, in both cases an undeliverable message is returned to sender. The only thing missing is the stamp. I guess this is where the analogy may break down a little. Still, although you probably don't pay specifically for sending an HTTP request to a server, you're probably paying for Internet service. So there's your postage stamp!

2.4.1 Namespaces

Understanding XML namespaces is important to understanding SOAP. I'm not going to cover all the details of XML namespaces, as that would take a long time and wouldn't reflect directly on the subject matter of this book. Nonetheless, I'll give you a quick description of XML namespaces as they relate to SOAP. Basically, namespaces are an XML mechanism for eliminating ambiguity between XML elements and attributes. In other words, they help us to understand the context, or meaning, of the element. Let's look at part of an earlier example:

```
<m:GetCurrentTemperature xmlns:m="WeatherStation">
       <m:scale>Celsius</m:scale>
</m:GetCurrentTemperature>
```

XML allows us to create something similar to programming language variables to represent namespaces. In the example above we've created a namespace identifier named `m` by using the namespace declaration `xmlns:m="WeatherStation"`. Generally, the syntax of a namespace declaration in a SOAP message typically is:

```
xmlns:id="some URI"
```

The `xmlns` keyword tells the XML processor that we're defining a namespace. If the definition includes a qualifier name, then we're also creating a namespace qualifier that can be used to distinguish this namespace from other namespaces. If we don't specify a qualifier name, then we're defining a default namespace. The `id` is the identifier name we're declaring, in this case, the letter `m`. The last part is a URI that provides the context for the namespace. Although the URI is frequently a URL, it's important to realize that it's just an identifier; the

[3] An exception to this occurs where an attachment is included. We'll look at that possibility in Chapter 8.

[4] There have been some discussions about providing for a `Body`-less Envelope in SOAP 1.2.

recipient of a document won't try to download the content of the URI (which may not even exist). We can use the letter `m` to qualify elements that are within the same scope as the declaration itself.

Qualifying an XML element means associating it with a specific namespace. In this example, we qualify the `GetCurrentTemperature` XML element with the namespace `WeatherStation` by declaring the element as `m:GetCurrentTemperature`. This means that `GetCurrentTemperature` is associated with the `WeatherStation` namespace, represented by the identifier `m`.

Note that the namespace identifier `m` can be used to qualify the element in which it is declared. But there's nothing special about using `m` in this element. We aren't required to qualify the `GetCurrentTemperature` element with the namespace identifier `m` just because that element contains the namespace declaration as an attribute, but rather because we need to indicate where `GetCurrentTemperature` is defined. Putting the declaration in this element creates the scope for which the identifier `m` can be used. Clearly, it's valid to use the identifier at the same level as its declaration; that's what we did in the example. So the scope of the namespace is bounded by the element that contains the declaration. This means that the namespace ID can be used on the element that contains the declaration, any attributes of that element, and any subelements and associated attributes of the containing element. The namespace identifier is not within the scope of the XML elements that are higher up in the containment hierarchy. If you go back and look at the original example, this means that the namespace identifier `m` could not be used for attributes of the `Body` element because `m` would be out of scope. The scoping rules are similar to those used in block-oriented programming languages like Java, where the namespaces are pushed onto a stack as you enter bracketed blocks of code, and the stack is popped as you exit the block. Here's a Java code snippet that shows the same kind of scoping for variables `var1` and `var2`:

```
int var1 = 10;
{
    int var2 = 2 * var1; // this is OK because var1 is in scope
}
var1 = var2; // this is invalid because var2 is out of scope
```

XML supports something called a *default namespace*, which can result in namespace qualification being inherited without being explicitly expressed. A default namespace is declared by assigning a value to the attribute named `xmlns` without using an associated namespace identifier. Consider this example:

```
<GetCurrentTemperature xmlns="WeatherStation">
    <scale>Celsius</scale>
</GetCurrentTemperature>
```

In this case, both the `GetCurrentTemperature` element and the `scale` element are associated with the `WeatherStation` namespace.

Namespace qualification is often necessary to determine the intended meaning of the element or attribute. Consider the following example:

```
<truckmonitor>
    <scale>37F6A</scale>
    <weight>12000</weight>
    <scale>Celcius</scale>
    <temperature>25</temperature>
</truckmonitor>
```

This XML contains some data that may have been collected at a truck monitoring station. The first occurrence of the `scale` element specifies the scale used to weigh the truck. Later on, we have the same element name, `scale`, to describe the temperature scale used to measure the outside temperature at the monitoring location. So how are we to know the meaning of the two scale elements? You might feel that the overall structure is intuitive just by looking at it, but this may not always be the case.

Certainly one way to avoid this problem would be to use more descriptive names, or XML structures that better represent the meaning of the elements. But you may not always be in control of those things; you may be creating a composite document from XML fragments that come from many different sources. The use of appropriate namespace qualifications can lend a hand in resolving name conflicts and ambiguity. The following document is the same as the previous one, with the two `scale` elements properly qualified:

```
<truckmonitor xmlns:ns1="TruckScale" xmlns:ns2="Thermometer">
    <ns1:scale>37F6A</ns1:scale>
    <weight>12000</weight>
    <ns2:scale>Celcius</ns2:scale>
    <temperature>25</temperature>
</truckmonitor>
```

The first occurrence of `scale` is now associated with the `TruckScale` namespace, while the second occurrence of `scale` is associated with the `Thermometer` namespace. Did you notice that both of the namespace declarations are attributes of the same element? That's perfectly valid, since namespace declarations are nothing more than attributes, and an XML element can have more than one attribute.

SOAP defines two namespaces to be used by SOAP messages. The SOAP `Envelope` is qualified by the namespace `http://schemas.xmlsoap.org/soap/envelope`. We used this namespace in earlier examples by declaring the `SOAP-ENV` namespace identifier and using it to qualify the `Envelope` element. The namespace identifier `http://schemas.xmlsoap.org/soap/encoding` is used to declare the `encodingStyle` attribute, which we'll discuss more later. Note that the `encodingStyle` attribute is namespace-qualified using the `SOAP-ENV` identifier. Here's a quick look at it again:

```
<SOAP-ENV:Envelope
          xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
          SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
         . . .
         . . .
</SOAP-ENV:Envelope>
```

2.5 The Envelope Element

The `Envelope` is the topmost element of the XML document that represents the SOAP message. The `Envelope` is the only XML element at the top level; the rest of the SOAP message resides within the `Envelope`, encoded as subelements. A SOAP message must have an `Envelope` element. The `Envelope` can have namespace declarations, as shown in the earlier examples, and needs to be qualified as shown earlier, using the `http://schemas.xmlsoap.org/soap/envelope` namespace. That's why the element name is shown as `SOAP-ENV:Envelope`. It is also common for the `Envelope` element to declare the `encodingStyle` attribute, with the attribute namespace-qualified using the declared namespace identifier `SOAP-ENV` as well.

All subelements and attributes of the `Envelope` must themselves be namespace-qualified. These elements and attributes are also qualified by `SOAP-ENV`, just as the `Envelope` is qualified. For the remainder of this chapter and the rest of the book, we'll use the `SOAP-ENV` namespace identifier to mean the `http://schemas.xmlsoap.org/soap/envelope` namespace. This should make for easier reading. Keep in mind that it's the namespace itself that matters, not the name used for the qualifier.

2.6 The Header Element

The SOAP header is optional. If it is present, it must be named `Header` and be the first subelement of the `Envelope` element. The `Header` element is also namespace-qualified using the `SOAP-ENV` identifier.

Most commonly, the `Header` entries are used to encode elements used for transaction processing, authentication, or other information related to the processing or routing of the message. This is useful because, as we'll see, the `Body` element is used for encoding the information that represents an RPC (or other) payload. The `Header` is an extension mechanism that allows any kind of information that lies outside the semantics of the message in the `Body`, but is nevertheless useful or even necessary for processing the message properly.

`Header` elements should limit the use of attributes to those elements that are immediate children of the `Header` element itself. The spec says that this *should* be done, meaning that although one can use them on elements deeper in the element hierarchy underneath the `Header` element, the recipient is required to ignore the attributes on such elements.

Here's an example of a SOAP header that contains an immediate child element named `username`. We don't apply any attributes to the `username` element, but we do namespace-qualify it. The `username` element identifies the user who is making the request.

```
<SOAP-ENV:Header>
    <ns1:username xmlns:ns1="JavaSoapBook">
      Jessica
    </ns1:username>
</SOAP-ENV:Header>
```

2.7 The actor Attribute

A SOAP message often passes through one or more intermediaries before being processed. For example, a SOAP proxy service may stand between a client application and the target SOAP service. We'll see an interesting example of this in Chapter 10, where we'll develop a SOAP application service that proxies another service on behalf of the client application, which actually specifies the ultimate service address. Therefore, the header may contain information intended solely for the intermediary as well as information intended for the ultimate destination. The `actor` attribute identifies (either implicitly or explicitly) the intended recipient of a `Header` element.

It's important to understand the requirements that SOAP puts on an intermediary. Essentially, it requires that any SOAP `Header` elements intended for use by an intermediary are not passed on to the next SOAP processor in the path. `Header` elements represent contracts between the sender and receiver. However, if the information contained in a `Header` element intended for an intermediary is also needed by a downstream server, the intermediary can insert the appropriate `Header` in the message to be sent downstream. In fact, the intermediary is free to add any `Header` elements it deems necessary.

If an `actor` attribute doesn't appear on a `Header` element, it's assumed that the element is intended for the ultimate recipient. In essence, this is equivalent to including the `actor` attribute with its value being the URI of the ultimate destination.

Let's extend the previous example by adding an `actor` attribute. Say that the message is being sent to an intermediate application server located at *http://www.mindstrm.com/AppServer*. We want that application server to log the name of the user that made the request, and then pass the request on to the final destination server. To do so, we set the `actor` for the `username` element to *http://www.mindstrm.com/AppServer*:

```
<SOAP-ENV:Header>
   <ns1:username
         xmlns:ns1="JavaSoapBook"
         SOAP-ENV:actor="http://www.mindstrm.com/AppServer">
         Jessica
    </ns1:username>
</SOAP-ENV:Header>
```

The proxy application server sees that the `username` element is intended for itself and not for the final destination, and uses the data appropriately. It removes the `username` element from the header before passing the message on to its ultimate destination. As you can see, the `actor` attribute is namespace-qualified by the `SOAP-ENV` identifier. That's because the `actor` attribute is defined by SOAP and specified by the associated `http://schemas.xmlsoap.org/soap/envelope` namespace.

2.8 The mustUnderstand Attribute

SOAP includes the concept of optional and mandatory header elements. This doesn't mean that the inclusion of the elements is mandatory — that is an application issue. Instead, "mandatory" means that the recipient is required to understand and make proper use of the information supplied by a `Header` element. This requirement allows us to accommodate situations in which a recipient of a SOAP message can't perform its job unless it knows what to do with the data provided by a specific `Header` element. In this case the element can include the `mustUnderstand` attribute with an assigned value of 1.

This may be necessary if the sending application is upgraded with a new version, for example. That new version may use some new information that has to be processed by the server in order for the result to be useful. Of course you'd expect that there would be a corresponding upgrade to the server, but maybe that hasn't happened yet. Because of the version mismatch, the older version of the server does not understand the new SOAP header element that it received from the upgraded client application. Let's say, for example, that the `username` header element must be understood by the recipient; if it is not, the message should be rejected. We can include this requirement in the SOAP message by assigning the `mustUnderstand` attribute the value of 1. (The value 0 is essentially equivalent to not supplying the attribute.) Let's modify our previous example to indicate that the recipient must understand the `username` element:

```
<SOAP-ENV:Header>
    <ns1:username
        xmlns:ns1="JavaSoapBook"
        SOAP-ENV:actor="http://www.mindstrm.com/AppServer"
        SOAP-ENV:mustUnderstand="1">
        Jessica
    </ns1:username>
</SOAP-ENV:Header>
```

If the recipient of this message does not understand the `username` element, it is required to respond with a SOAP fault. The response might look something like this:

```
HTTP/1.0 500 Internal Server Error
Content-Type: text/xml; charset="utf-8"
Content-Length: 373

<SOAP-ENV:Envelope
          xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   <SOAP-ENV:Body>
      <SOAP-ENV:Fault>
         <faultcode>SOAP-ENV:MustUnderstand</faultcode>
         <faultstring>SOAP Must Understand Error</faultstring>
         <faultactor>http://www.mindstrm.com/AppServer</faultactor>
      </SOAP-ENV:Fault>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The fault response includes a `faultcode`, a `faultstring`, and a `faultactor`; the `faultactor` indicates where the fault took place. We'll look more closely at faults later in this chapter.

2.9 The encodingStyle Attribute

The `encodingStyle` attribute specifies the data encoding rules used to serialize and deserialize data elements. It is important to understand that SOAP does not specify any default rules for data serialization. SOAP does, however, specify a simple data typing scheme commonly supported by SOAP implementations. This is the subject of the next chapter, so we won't get into the details of the encoding rules here. However, it's important to understand how to specify the encoding style to be used for serializing and deserializing the element data in the SOAP message.

The `encodingStyle` attribute is namespace-qualified using the `SOAP-ENV` namespace identifier. In the following example we specify the `encodingStyle` attribute as part of the `Envelope` element; we'll see examples in later chapters that make this declaration in the `Body` element instead. Either way works, as long as you recognize that the `encodingStyle` attribute applies to the element in which it was declared as well as all of its subelements (i.e., it's in-scope).

```
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    . . .
    . . .
</SOAP-ENV:Envelope>
```

You can supply more than one URI in the declaration. In that case, the first URI must be the most specific, and the last must be the least specific. Here's an example:

```
SOAP-ENV:encodingStyle="http://www.mindstrm.com/tightEncoding
http://www.mindstrm.com/looseEncoding"
```

In this case, the system looks for the encoding rules first using *http://www.mindstrm.com/tightEncoding*.[5] If the rule can't be found using that URI, the system then tries *http://www.mindstrm.com/looseEncoding*. It's possible to turn off the currently scoped encoding style by specifying an empty string for the URI (`""`). This declaration applies to the current element and all of its subelements.

2.10 Envelope Versioning

The SOAP spec doesn't define a numbering system for declaring which version of the SOAP envelope you're using. In SOAP 1.1, all envelopes must be associated with the `http://schemas.xmlsoap.org/soap/envelope` namespace,[6] which we've used in all of the previous examples. That's it. No other versioning information is used for SOAP envelopes at this time. If a SOAP message is associated with any other namespace, or if no namespace is declared, then the recipient has to respond with a SOAP version mismatch. Here is an example of a SOAP fault response for this situation:

[5] It's tempting to think of the system trying to download `tightEncoding`, particularly since the URI happens to be a URL. But that isn't the case; the system just looks in its own tables to see whether it understands `tightEncoding`, and acts accordingly.

[6] For SOAP 1.2, the namespace is http://www.w3.org/2001/12/soap-envelope.

```
HTTP/1.0 500 Internal Server Error
Content-Type: text/xml; charset="utf-8"
Content-Length: 311

<SOAP-ENV:Envelope
          xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   <SOAP-ENV:Body>
      <SOAP-ENV:Fault>
         <faultcode>SOAP-ENV:VersionMismatch</faultcode>
         <faultstring>SOAP Envelope Version Mismatch</faultstring>
      </SOAP-ENV:Fault>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

2.11 The Body Element

The SOAP `Body` element is mandatory in SOAP 1.1. If there is no SOAP header, the `Body` must be the first subelement of the `Envelope`; otherwise it must directly follow the SOAP `Header` element. The `Body` element is also namespace-qualified using the `SOAP-ENV` identifier.

The `Body` element contains the SOAP request or response. This is where you might find an RPC-style message that contains the method name and its parameters, or a one-way message and its relevant parts, or a fault and its details. SOAP `Body` elements are not completely defined by the SOAP specification; we'll cover that subject in great detail later. In fact, SOAP defines only one kind of `Body`: the SOAP `Fault`.

Let's take another look at one of the earlier examples of an envelope with a SOAP `Body` element. In this case the request is an RPC. We've namespace-qualified the `Body` element using the same `SOAP-ENV` namespace identifier that we used for the other SOAP-defined elements and attributes.

```
<SOAP-ENV:Envelope
          xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
          SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
         <m:GetCurrentTemperature xmlns:m="WeatherStation">
                <m:scale>Celsius</m:scale>
         </m:GetCurrentTemperature>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

2.12 SOAP Faults

The `Fault` is the only `Body` element entry defined by SOAP. It's used to carry error information back to the originator of a SOAP message. The `Fault` element must appear as an immediate subelement of the `Body` element, and it can't appear more than once.

SOAP defines four subelements of the `Fault` element. Not all of these are required, and some are appropriate only under certain conditions. These elements are described in the following sections. You'll probably recognize that we've seen all of these in previous examples, so you may want to flip back and look at them again after you've read these descriptions.

2.12.1 The faultcode Element

The `faultcode` element provides an indication of the fault that is recognizable by a software process, providing an opportunity for the system receiving the fault to act appropriately. The code is not necessarily useful to humans — that is the purpose of the `faultstring` element described in the next section. The `faultcode` element is mandatory, and must appear as a subelement of the `Fault` element. SOAP defines a number of fault codes for use in the `faultcode` element. These codes are associated with the `http://schemas.xmlsoap.org/soap/envelope` namespace. Here are brief descriptions of the SOAP-defined fault codes:

`VersionMismatch`

> Indicates that an invalid namespace was associated with the SOAP `Envelope`.

`MustUnderstand`

> Means that there was a SOAP `Header` element that contained the `mustUnderstand` attribute with a value of 1, and either the attribute was not understood or the semantics associated with the attribute couldn't be followed.

`Client`

> Indicates that there was some error in the formatting of the SOAP message or that the message did not contain the appropriate or necessary information. The client should assume that the message is not suitable to be sent again without making the appropriate changes. We saw an example of this when we requested the local temperature using an invalid temperature scale.

`Server`

> Indicates that the message could not be processed due to reasons not related to the formatting or contents of the received message. This can be interpreted to mean that the message could be re-sent without modification possibly processed at some later time. For example, the back-end database server needed to complete the requested action may be currently offline, but may be back online later.

These fault codes are extensible; this means that they can be extended using a dot notation, where the name following each dot serves to provide more specific detail. For example:

```
Server.WeatherStationFailure
```

While this can still be considered a `Server` error, it provides more information by indicating that there was some problem with the weather station equipment needed to fulfill the request. We could extend this fault code even further, as follows:

```
Server.WeatherStationFailure.SmashedToPieces
```

This fault code lets us know that the weather station equipment is beyond repair, possibly because it fell off the building and was subsequently crushed. Remember, however, that these codes are meant to be processed by software. Extended fault codes should be defined in anticipation of possible situations or conditions, allowing the recipient of the fault to act accordingly. (I'm not sure that smashed equipment would qualify as an anticipated condition.)

2.12.2 The faultstring Element

This element is also mandatory, and must appear as a subelement of the `Fault` element. Its purpose is to provide a human-readable description of the fault; it's not designed to be processed by the recipient the way the `faultcode` element is. It is expected that the sender will populate this element with a reasonable description that would make sense to the reader within the context of the original request.

2.12.3 The faultactor Element

This element is required only under certain conditions, and when present must be a subelement of the `Fault` element. It is the corollary to the `actor` header attribute described earlier. It provides an indication of the system that was responsible for the fault. Earlier, we talked about SOAP intermediaries and used a proxy application server as an example. There, the proxy generated the fault because it didn't understand what to do with the `username` element. Because the application proxy was not the intended ultimate recipient of the original message, it was required to include the `faultactor` element, identifying itself as the source of the fault. The value of this element is the URI of the fault generator.

If the source of the fault is the ultimate destination of the message, the `Fault` element is not required to include a `faultactor` element. However, this element may be included even under these circumstances.

2.12.4 The detail Element

The `detail` element provides information related to faults that occur due to errors associated with the `Body` element of the request message. If the contents (or lack of contents) of the `Body` preclude the proper processing of the message, then the `detail` element must appear as a subelement of the `Fault`. On the other hand, if the fault is not related to the `Body` element of the message, the `detail` element must not be included; SOAP specifies that the absence of a `detail` element indicates that the fault is unrelated to the processing of the `Body` element.

In particular, the `detail` element cannot be used to further describe faults related to SOAP `Header` elements. In the earlier example, the proxy application server was not able to further describe the fault using a `detail` element because the problem was unrelated to the contents of the `Body`.

2.12.5 Another Fault Example

Here's another complete example of a SOAP fault:

```
HTTP/1.0 500 Internal Server Error
Content-Type: text/xml; charset="utf-8"
Content-Length: 595

<SOAP-ENV:Envelope
          xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   <SOAP-ENV:Body>
      <SOAP-ENV:Fault>
         <faultcode>SOAP-ENV:Client</faultcode>
         <faultstring>Client Error</faultstring>
         <faultactor>http://www.mindstrm.com/LocalWeather</faultactor>
         <detail>
            <m:weatherfaultdetails xmlns:m="WeatherStation">
               <message>No such temperature scale: Calcium</message>
               <errorcode>1234</errorcode>
            </m:weatherfaultdetails>
         </detail>
      </SOAP-ENV:Fault>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

This is really the same response we used earlier in the chapter. The only difference is that this time we included the `faultactor` element to identify the source of the fault. So this example includes an instance of every SOAP fault code, as well as a `detail` element describing the fault a little further.

At this point, we're finished looking at the SOAP message itself. Our next step is to look at the encodings that are used in messages.